Hardening a Real-Time Operating System for a Dependable RISC-V System-on-Chip

Benjamin Mezger, Douglas Santos, Luigi Dilillo, Douglas Melo

Title: Hardening a Real-Time Operating System for a Dependable RISC-V System-on-Chip

Author(s): Benjamin W. Mezger, Douglas A. Santos, Douglas R. Melo, Luigi Dilillo

Document version: Post-print version (Final draft)

Please cite the original version:
B. W. Mezger *et al.*, "Hardening a Real-Time Operating System for a Dependable RISC-V System-on-Chip," 2023 36th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2023.

Hardening a Real-Time Operating System for a Dependable RISC-V System-on-Chip

Benjamin W. Mezger*, Douglas A. Santos†, Luigi Dilillo†, and Douglas R. Melo*
*LEDS, University of Vale do Itajaí, Itajaí, Brazil
†IES, University of Montpellier, CNRS, Montpellier, France
ben@edu.univali.br, {douglas.santos, luigi.dilillo}@umontpellier.fr, drm@univali.br

Abstract

In safety-critical systems, solutions that rely on redundancies at the hardware and software levels allow these systems to operate in radiation-harsh environments. In the literature, software-based techniques for enhancing reliability in critical systems are less developed when compared to hardware-based fault-tolerant techniques. In this context, we implemented error correction and detection techniques through software, along with a custom RISC-V processor-core implementation. RISC-V is an open instruction set architecture that is gaining popularity due to its modular design, as well as the FreeRTOS kernel, which is a well-established Real-Time Operating System. Our implemented proposal has shown to be able to reduce the number of execution failures by four times when compared to the standard version at the cost of increased execution time and energy consumption. The experimental results may guide designers in choosing the best trade-offs between reliability and resource constraints in complex contexts such as space applications.

Index Terms

Systems-on-Chip, Fault Tolerance, Software Reliability, RISC-V, FreeRTOS

I. Introduction

RISC-V is an open standard architecture based on the Reduced Instruction Set Computer (RISC) principles, initially developed by Berkeley University but maintained by the RISC-V foundation. RISC-V provides a base instruction set with several optional extensions, reducing software costs and enabling implementations of large- and small-scale architectures [1]. Semico Research Corp. estimates that in 2025, the market will have around 62.4 billion RISC-V cores worldwide [2].

RISC-V's software support is increasing in multiple computing fields, such as Operating System (OS), on which the Linux kernel [3] and Zephyr [4] already have support, as well as compilers and simulation software. In addition to a general-purpose OS, systems with additional characteristics, such as real-time systems provided by FreeRTOS [5] are also available. However, approaches focusing on delivering reliability are not near the same number in RISC-V.

An operating system may run in environments that demand reliability techniques. These systems can experience external interference, impacting the system's availability, security, integrity, or maintainability [6]. For these systems to operate correctly in the presence of a fault, the system may provide reliability at the hardware or software levels.

Several works in the literature tackle reliability characteristics at the software level, such as the work [7], where the authors inject single bit-flips to simulate faulty behavior in the RISC-V multiplier unit. The work [8] implements reliability techniques for software by adding fault mitigations to instructions and/or data at the compiler level. The work [9] proposes the simulation of errors in a set of RISC-V compiled C programs to evaluate the impact of software execution when injecting permanent and transient faults. Moreover, the work [10] evaluates the impact of injecting single bit-flips in the configuration memory in a Linux-based system. Finally, [11] analyses the pros and cons of hardened-by-replication software applications running on soft-core microprocessors.

Solutions explore redundancy or rely on fault-tolerant hardware components to guarantee reliability for safety-critical applications. Hardened hardware components are not always cost-effective, nor do they always provide the specificity a system needs. Therefore, this work explores software-based reliability techniques and proposes applying a technique at OS level on a RISC-V architecture. Furthermore, we propose a set of modifications of a RISC-V processor to monitor and communicate the OS of a detected fault.

Given this context, we port FreeRTOS to a reliable RISC-V processor and implement the required exceptions and interrupts for supporting the OS. We implement process-level Triple Module Redundancy (TMR), which blocks the running tasks and waits until all task values have been compared. Further, we implement fault notification through a custom RISC-V exception to allow the OS to log, reset and do any required post-processing when a fault is detected.

This work was supported in part by the Foundation for Support of Research and Innovation, Santa Catarina (FAPESC-2021TR001907), and the Brazilian National Council for Scientific and Technological Development (CNPq - process 350794/2023-5), and the Region d'Occitanie and the École Doctorale I2S from the University of Montpellier (contract 20007368/ALDOCT-000932).

II. Background

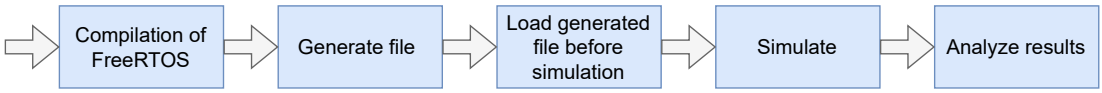

Fig. 1. HARV and FreeRTOS compilation and simulation process.

HArdened RISC-V (HARV) [12] is a low-cost fault-tolerant, single-cycle RISC-V processor based on the RV32I instruction set of the non-privileged specification. The HARV processor is described in VHDL without any vendor-specific Intellectual Property (IP) blocks. Furthermore, this processor has been thoroughly analyzed in irradiation facilities and provides detailed information on radiation-induced errors [13]. Until now, HARV was only used in specific and dedicated applications running in bare-metal.

FreeRTOS is a real-time OS for microcontrollers designed to execute user-defined tasks within a strict period. FreeRTOS's architecture is structured into three main layers: (*i*) the tasks layer, which deals with creating user-defined tasks and scheduling maintenance, (*ii*) the communication layer, allowing tasks and interrupts to send data to each other and signal resources, and (*iii*), the shim layer, which acts between the hardware-independent and -dependent code [14].

Since HARV did not provide support to any OS, we ported FreeRTOS to increase its software support. We used the existing FreeRTOS RISC-V port available in the main repository and modified the source code to adapt to HARV's current architecture. The RISC-V port facilitated the modification and validation of the necessary parts of the source code while hiding details that were not the main focus of this work.

To simulate the compiled binary and validate our port, we ran a script against the binary file to convert it from a binary to a text with the instruction format and data in hexadecimal format. The simulation then parses all lines of the generated file and executes the instructions. This flow is illustrated in Fig. 1.

Furthermore, given that HARV implements the RV32I instruction set, this work used the open-source FreeRTOS RISC-V implementation [15] as the baseline of the port to make the necessary modifications.

III. Operating System Hardening

To add support to FreeRTOS in the HARV architecture, modifications were made to support basic functionalities of the FreeRTOS kernel, both in hardware (HARV) and in software (FreeRTOS). We identified three structures of Amazon's FreeRTOS port that required changes to adapt to the HARV processor:

1) The `vPortSetupTimerInterrupt` to set up the timer interrupt;
2) `portcontextSAVE_INTERRUPT_CONTEXT` to handle a context switch and save the current context;
3) The linker script provided by FreeRTOS source demo for the linking phase.

The FreeRTOS `vPortSetupTimerInterrupt` function required changes on the `mtime` register, given that in RISC-V specification, it is memory-mapped and in HARV is an actual Control Status Register (CSR). A change to `portcontextSAVE_INTERRUPT_CONTEXT` procedure was required to support the `mstatus` CSR, given that `mstatus` was not recognized by the compiler during the compilation phase, which required us to change the CSR register to its correct address in HARV. Finally, the linker script required CSR, and Read-Only Memory (ROM) addresses modification to find the initial address of the program.

Our findings later showed that Amazon's RISC-V version of FreeRTOS distribution was out of date along with the FreeRTOS kernel repository. The changes were later implemented along with refactoring on trap handling and context switching[1]. To overcome such problems, we copied the new refactor of the FreeRTOS Kernel source code to the FreeRTOS distribution source. This source code replacement was enough to get HARV to execute the demo software provided by FreeRTOS.

While the RISC-V specification defines a range of exception codes, HARV had no support for software traps and timer interrupts. Four traps, shown in Table I, were identified to be relevant to HARV to support FreeRTOS.

To implement the trap support, a new set of CSRs is required, including (*i*) `mie` CSR for read/write of interrupt enable bits, (*ii*) memory-mapped `mtime` real-time cycle counter, (*iii*) `mcause` for indicating the event that caused a trap and (*iv*) `mtval` for assisting software with exception-specific information. The implemented CSRs, and their privilege level are shown in Table II.

The new trap encoder unit is connected to the Instruction Fetch (IF), Control, and ALU unit. The trap encoder handles timer interrupt by comparing if the current timer (`mtime`) value is greater or equal to the value set in

[1]Pull Request can be found at https://github.com/FreeRTOS/FreeRTOS-Kernel/pull/444

TABLE I
EXCEPTIONS AND INTERRUPTS IMPLEMENTED.

Exception type	Exception name
Synchronous	Environment call from Machine-Mode
Synchronous	Load access fault
Synchronous	Store access fault
Asynchronous	Machine timer interrupt

TABLE II
IMPLEMENTED CSRS IN HARV.

Number	Privilege	Name	Description
0x304	Machine R/W	`mie`	Interrupt Enable
0x342	Machine R/W	`mcause`	Trap Cause
0x343	Machine R/W	`mtval`	Trap Value
Memory-mapped	Machine R/W	`mtime` and `mtimecmp`	Timer

`mtimecmp`. Furthermore, the trap encoder writes any pending interrupts before encoding the trap matching any exception shown in Table I and writing the `mcause` if applicable. The CSR components were extended to support trap-related CSR. Fig. 2 illustrate the HARV organization after the inclusion of the trap encoder.

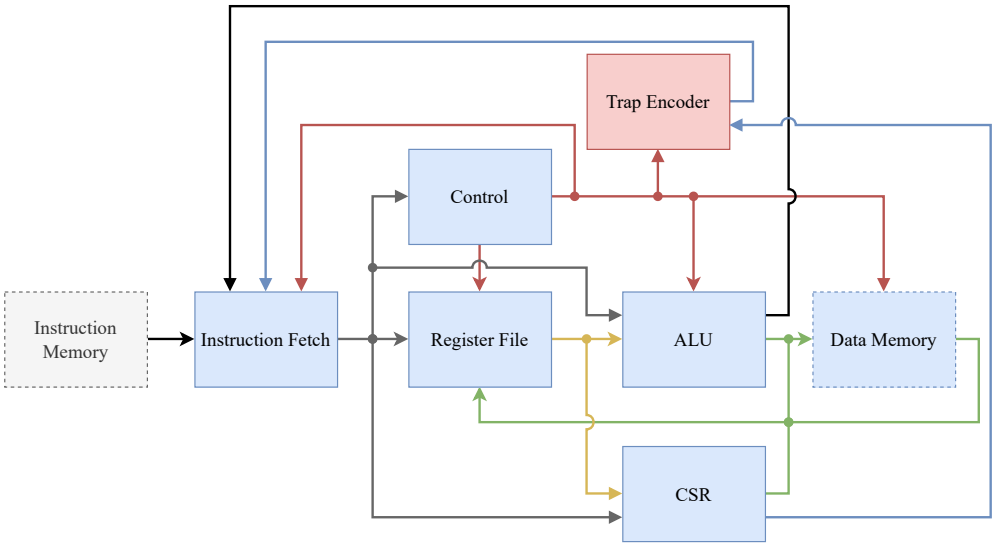

Fig. 2. HARV organization with trap support.

A. Fault trap support

Given the HARV processor has support for fault detection and correction through means of TMR and Hamming, notifying the OS of a fault enables handling fault mitigation at the software level or running any post-processing mechanism. Post-processing mechanisms can include logging fault information, fault correction, or running safety checks to decide whether the system should halt or continue running.

To support fault mitigation in HARV at the software level, we add support to a new RISC-V custom asynchronous exception. We use the exception code `18` because it is designated for platform use. We modify HARV's Hamming and TMR component to raise an asynchronous exception and set `mcause` to value `18` to notify the OS of a hardware-detected fault. Since not all designs require trapping on every fault, we can enable or disable this feature during compilation by defining the `FAULT_TRAP_HANDLER` flag with the function's name.

B. Software-based fault tolerance

To further harden the system, we applied TMR to correct errors based on different task output results. In our proposal, if the process has no redundancy mechanism, we run the process and continue without any modification.

Otherwise, our task-based TMR runs a copy of three processes with independent memory space and then compares the output of all three tasks byte-by-byte. We run an optionally provided post-processing mechanism if a fault is detected during the voting procedure.

To synchronize the output of the tasks, a private queue is used to store the computed values. Each task replica is responsible for inserting the shared value into the queue and blocking it until all other tasks have inserted their value. When all processes have synchronized, we compare the values inserted in the queue byte-by-byte and trigger an optional exception handler if two values do not match. Further, our solution ensures that the value address is within the memory range if a fault changes the address value to an out-of-range address.

Fig. 3 shows a flowchart of the task's synchronization and voting procedures, including inserting a value into the queue, blocking, and context switching to another task. Finally, when the queue is full, we compare the results and trigger a fault trap if enabled by the user and unblock any blocked tasks waiting for the result.

TABLE III
IMPLEMENTED FREERTOS API.

Interface	Description
`vTmrInit(void (*)(void *), ...)`	Registers and enable tasks for TMR.
`int iTmrInsertValue(void (*)(void *), void *addr, int size)`	Insert value in the queue for TMR comparison. Receives the task pointer, value address, and value size.
`int iTmrPullData()`	Check if queue is full. Returns `0` if queue is empty, `1` otherwise.
`void ft_exception_handler(void *arg)`	Optional interface for implementing a custom exception handler in case an error if detected during the voting process.
`void vPrintTask()`	Print list of current registered tasks.
`void vTmrWaitForData()`	Force blocking the current task until all TMR tasks have ran.
`void fault_trap_handler( unsigned long, unsigned long, int, int)`	Required interface for enabling fault trap notification. It receives `mcause`, `mepc`, `event_id` and `mem_event_id` as argument.

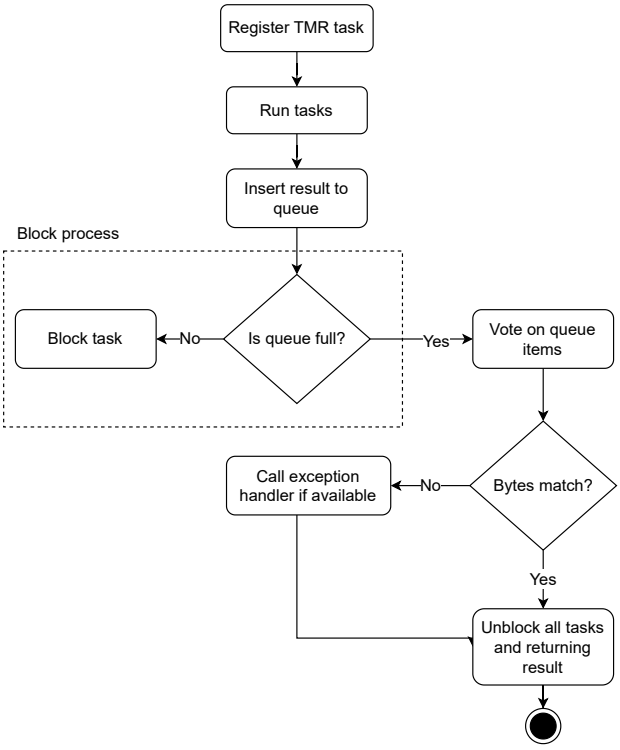

Fig. 3. Task-level TMR flow.

The TMR functionality further provides *N*-version support by enabling the programmer to specify different versions of the function for redundant execution. With support for *N*-version functions, one can benefit from TMR by running the

function three times while having three different versions of the implementation. The implementation provides a public Application Programming Interface (API), shown in Table III, with functions and constants for users to synchronize tasks and check for faults affecting data.

IV. Results

This section presents the results related to hardware and software cost, performance, and dependability obtained through the fault injection campaign.

A. Cost

We used Xilinx Vivado 2021.1 to synthesize the HARV processor targeting the Xilinx 7000 FPGA. Table IV presents the cost in terms of hardware resources for the synthesis of the HARV before and after the addition of the trap encoder. The hardware upgrade increased 56.31% in the number of FFs and led to an increase of 46.03% in LUTs compared to the baseline, which ended up degrading the operational frequency in 0.8%.

TABLE IV
HARV SYNTHESIS RESULTS.

HARV	LUTs	FFs	Fmax (MHz)	Ptotal (mW)
Standard	3117	1893	51.59	222
With trap support	4552	2959	51.18	237

To analyze software resources, we used FreeRTOS standard toolchain (GNU GCC) to compile and evaluate binary segment size. Table V presents the cost of binary size (in bytes) for the standard and hardened FreeRTOS versions. We had an increase of 3.78% in terms of code (section `.text`) and an increase of 18.87% in read-only data (`.rodata`). Considering the total size of the binary, the hardened version of the FreeRTOS presents an increase of 1.90% in the segment with respect to the standard version.

TABLE V
BINARY SIZE OF FREERTOS STANDARD AND HARDENED.

Segment (in bytes)	**FreeRTOS standard**	**FreeRTOS hardened**
`.text`	69880	72524
`.rodata`	1992	2368
`.data`	28	28
`.bss`	73188	73188
`.stack`	12284	12284
Total	158164	161184

B. Performance

We used Xilinx Vivado's simulator to evaluate the modified HARV in terms of performance for standard and hardened versions of FreeRTOS. The simulation gave as result the total number of cycles for each execution. Taking into account the frequency at which HARV operates (detailed in Table IV), the performance results in Table VI show that the total execution time for the hardened increases approximately two times compared to the standard FreeRTOS version. A similar analysis can be made for energy consumption, considering the total power presented in Table IV.

TABLE VI
PERFORMANCE RESULTS.

FreeRTOS	Cycles	Execution time (s)	Energy (mJ)
Standard	8.54×10^7	1.61	380.95
Hardened	1.73×10^8	3.27	771.24

Table VII presents the performance overhead of cycles introduced by each TMR function to provide fault tolerance. For this measurement, the cycle count is fetched before and after entering the function call.

TABLE VII
CYCLE COUNT PER API FUNCTION WITH TMR ENABLED.

Function	Cycle start	Cycle end	Total
`vTmrInit`	1,583,504	1,854,083	270,579
`iTmrInsertValue`	6,478,322	11,992,678	5,514,356
`vTmrCompare`	7,072,730	11,783,849	4,711,119

The standard version does not execute `vTmrInit` and `vTmrCompare` functions, as it does not initialize any TMR tasks, and neither calls the TMR compare to check all task values. The call to `iTmrInsertValue` has the highest number of cycles, given it blocks the process until all values are fed to the queue and compared, thus blocking each process at each call in case the queue is not filled. Further, the `vTmrCompare` function is dependent on the size of the value that is being compared, while `iTmrInsertValue` is dependent on the number of cycles `vTmrCompare` requires.

C. Experimental Setup

To simulate the fault occurrence for evaluating the effectiveness of our software-based hardening proposal, we made a fault injection campaign that inserts Single-Event Upset (SEU) faults in random flip-flops at random times. We used Intel ModelSim SE 2019.01 to simulate faults through a TCL script. To run the simulations, we used a server running CentOS, with 512 gigabytes RAM and a 96-core Intel (Haswell) CPU, with a 64-bit architecture.

The first step of the simulation is the execution of a golden run, which executes once the entire application without fault injections. This run provides baseline indicators for the comparison with the subsequent simulations, i.e., execution time, system flip-flops, and UART output. These indicators are also used as parameters for the fault injections.

The fault injection is based on a neutron-characterized flip-flop FIT_{NYC} (Failure In Time for a billion hours in the New York City's neutron flux at sea level) of 248 and a neutron flux of $5 \times 10^{10} n/cm^2/s$. The computation of the fault injections is presented in Fig. 4, in which the script steps over the execution time in random increments calculating a fluence estimation for each time delta. We use the total fluence to calculate the error rate for each flip-flop during this period, which uses uniform randomization to decide whether the fault should be injected.

Regarding workload, we implemented three FreeRTOS tasks that iterate until a given number and multiply the number by 2 through adder, multiplier, and bit-shifting. Since the tasks that we implemented are simple operations, instead of keeping the result of the multiplication stored in a variable, we iterated through an array of `512` positions and store the generated value in all array positions, to simulate an intensive computational operation and increase the chances of getting an error propagation.

D. Fault injection campaign

We ran each simulation with a set of 1000 executions. Each execution creates a `uart.log` log file with all the output and the simulation time that were next used to analyze the simulation results. At the end of all 1000 executions, we ran a Python script for all the log files to analyze the results obtained for each simulation. The Python script verifies which execution finished correctly and which did not and how many faults were detected and corrected by the software.

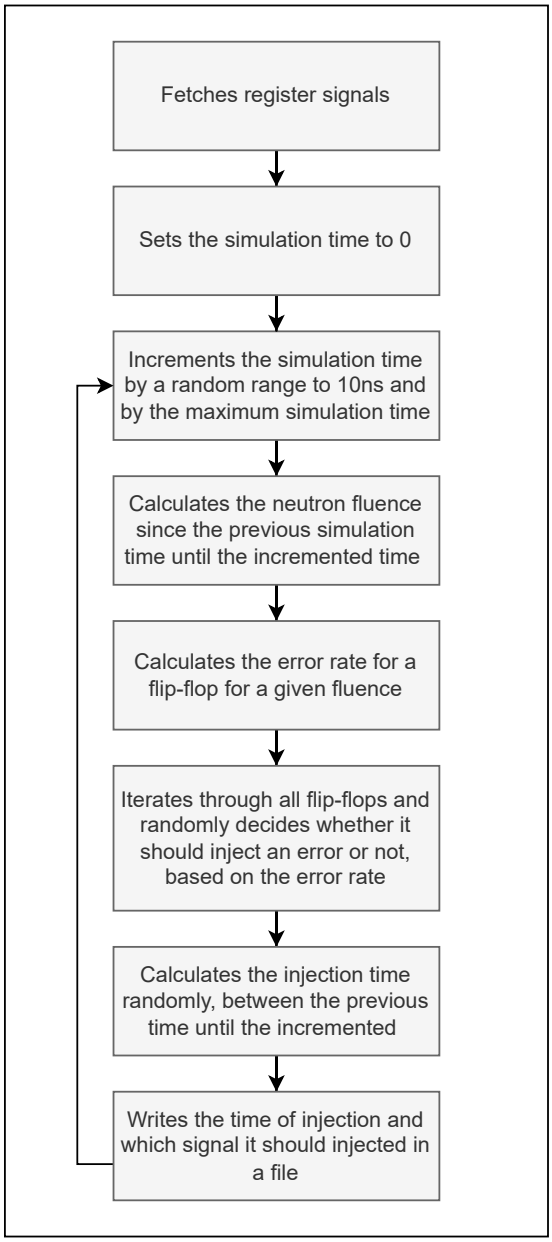

Fig. 4. Fault injection flow.

A full simulation (1000 executions) took 120-150 hours to complete. In total, the simulations took around 600 hours (25 days). The results of all simulations are shown in Table VIII.

TABLE VIII
FAULT INJECTION CAMPAIGN RESULTS.

Classification	FreeRTOS Standard	FreeRTOS Hardened
Number of executions	1000	1000
Executions with error	694	933
Executions with errors detected	N/A	765
Executions with errors corrected	N/A	22
Executions with failure	694	168

Of the 1000 executions for each simulation, we observed the standard version had 694 executions with errors, while the hardened version had 933 executions with errors and an increase of 34.43%. This is because the hardened

version takes approximately two times more for its execution in comparison to the standard version, exposing the system to more errors compared to the standard version in the simulation. Considering the detected errors, the hardened FreeRTOS detected 765 errors from 933 executions with errors. Among the 765 executions with errors, 22 errors were not only detected but also corrected.

Considering the reliability analysis, the standard version had 694 executions with errors propagated into failure because of the absence of detection and detection mechanisms. The hardened FreeRTOS presented only 16.8% of executions with failure, in which the system was not able to neither detect nor correct the error. Despite the execution taking twice as long exposing the system to twice the number of injected errors, the experiments show that the hardened version of FreeRTOS was able to reduce the number of execution failures by 4 times, since it was able to detect the error before it would provoke a failure.

E. Discussion

To achieve the goals of this work, modifications to the processor and OS were necessary. These modifications had a significant increase in the logical resources and a small increase in terms of binary size when compiling the OS with hardened features.

Concerning performance, we noticed the execution time for the hardened version doubled. We also observed that the execution time increase is mainly due to the TMR usage, which blocks all tasks until it finishes executing. Moreover, the input size that will be compared by the TMR module will impact the execution time since the module will compare byte-by-byte of each input.

Concerning reliability, we made a fault injection campaign that randomly inserted SEUs. We ran 1000 executions for each simulation, in which we observed that the hardened version presented more executions with errors due to the increase in execution time. Besides this drawback, the hardened FreeRTOS presented a reduction of the number of executions failure by four times. In applications where the main requirement is reliability, this hardened version presents a good trade-off between execution time and fault tolerance.

V. Conclusion

This work aimed to provide a software-level hardening solution using FreeRTOS in a RISC-V architecture. For this purpose, we modified the HARV processor core to provide trap handling and fault notification through a custom exception. Further, we added TMR support for FreeRTOS tasks, including registering a fault trap handler in case the HARV detected a fault.

The experiments showed the hardened FreeRTOS capability to reduce executions failure four times lower than the standard one. This performance was obtained at the price of increased energy consumption and execution time. This trade-off can be considered suitable for applications where reliability has a higher priority than other metrics.

In future work, we intend to submit the proposed implementation to a fault injection campaign in particle accelerators and compare fault-tolerant techniques at the hardware level and techniques implemented at the software level.

REFERENCES

[1] B. W. Mezger, D. A. Santos, L. Dilillo, C. A. Zeferino, and D. R. Melo, "A survey of the risc-v architecture software support," *IEEE Access*, vol. 10, pp. 51 394–51 411, 2022.

[2] J. Osier Mixon, "Semico Forecasts Strong Growth for RISC-V," November 2019. [Online]. Available: https://riscv.org/announcements/2019/11/9679/

[3] L. Torvalds, "RISC-V Port for Linux 4.15 v9," November 2017. [Online]. Available: https://lkml.org/lkml/2017/11/15/640

[4] C. Lockwood, "Zephyr Project Blog Post: Zephyr RTOS Featured In RISC-V Getting Started Guide," June 2019. [Online]. Available: https://riscv.org/news/2019/06/zephyr-project-blog-post-zephyr-rtos-featured-in-risc-v-getting-started-guide/

[5] J. Barr, "RISC-V Support in the FreeRTOS Kernel," February 2019. [Online]. Available: https://aws.amazon.com/blogs/aws/new-risc-v-support-for-freertos-kernel/

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[7] J. Laurent, V. Beroulle, C. Deleuze, and F. Pebay-Peyroula, "Fault injection on hidden registers in a risc-v rocket processor and software countermeasures," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 252–255.

[8] M. Bohman, B. James, M. J. Wirthlin, H. Quinn, and J. Goeders, "Microcontroller compiler-assisted software fault tolerance," *IEEE Transactions on Nuclear Science*, vol. 66, no. 1, pp. 223–232, 2018.

[9] P. Adelt, B. Koppelmann, W. Mueller, and C. Scheytt, "A scalable platform for qemu based fault effect analysis for risc-v hardware architectures," in *MBMV 2020-Methods and Description Languages for Modelling and Verification of Circuits and Systems; GMM/ITG/GI-Workshop*. VDE, 2020, pp. 1–8.

[10] I. Wali, A. Sánchez-Macián, A. Ramos, and J. A. Maestro, "Analyzing the impact of the operating system on the reliability of a risc-v fpga implementation," in *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2020, pp. 1–4.

[11] C. De Sio, S. Azimi, A. Portaluri, and L. Sterpone, "Seu evaluation of hardened-by-replication software in risc-v soft processor," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2021, pp. 1–6.

[12] D. A. Santos, L. M. Luza, L. Dilillo, C. A. Zeferino, and D. R. Melo, "Reliability analysis of a fault-tolerant risc-v system-on-chip," *Microelectronics Reliability*, vol. 125, p. 114346, 2021.

[13] D. A. Santos, A. M. Mattos, D. R. Melo, and L. Dilillo, "Enhancing fault awareness and reliability of a fault-tolerant risc-v system-on-chip," *Electronics*, vol. 12, no. 12, p. 2557, 2023.

[14] A. Brown and G. Wilson, *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. Lulu.com, 2011, vol. 1.

[15] Amazon, "FreeRTOS Kernel," https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/main/portable/GCC/RISC-V, 2022.